Name: Mohsin Jabbar Tunio

Roll no: 2k18/csee/54

Assigned by: Sir Gulsher Laghari

Department: Computer science Evening

Paper 2

Title And Authors:-

**Forking Without Clicking: on How to Identify Software Repository Forks**

* Antoine Pietri
* Guillaume Rousseau
* Stefano Zacchiroli

Conference Program MSR 2020

Introduction And Motivation:-

How developers and software communities work on their projects, and how this relationship evolves over time, have been topics of interest in software engineering research for many decades. The first drawback of trusting platform metadata as source of truth for what repository is a fork is that it is platform-specific. One cannot identify as forks repositories hosted on GitHub that has been forked from, say, GitLab, or more generally non-GitHub hosted repositories, and vice-versa. Similarly, although arguably less relevant from a quantitative point of view, one cannot recognize as forks, say, Git repositories used to collaborate with Subversion repositories via git-svn. For a fork ecosystem to be properly studied via the current approach, all the parallel development must happen using the same VCS and on the same platform. While the prevalence of Git does not seem to be waning, Git code hosting diversity is increasing, making the platform-specific part of this problem potentially severe. A second, more subtle methodological drawback is that trusting platform metadata introduces a selection bias on both the amount and type of forks that are considered. The fact that social coding platform strongly encourage, and sometimes even automate, the creation of forked repositories as the main way to contribute even the smallest one-liner change, inflates the number of forks. Many of these (soft) forks will be short-lived in terms of development activity. Hard forks will comparatively be more long lived and will not necessarily reside on the same code hosting platform. The example of the Linux kernel community is revealing in this respect: several copies of the full development history of Linux exist on GitHub, but are not recognizable as forks of torvalds/linux according to platform metadata, because kernel development does not primarily happen on GitHub and kernel developers create their repositories using git clone.

Research And Methodology:-

WHAT IS A FORK?

In this section we explore the spectrum of possible definitions of what constitutes a fork. In the following we will use the term “fork” to mean a forked software repository, without discriminating between “hostile” (or hard forks, according to the terminology of [36]) and development forks. We propose three definitions, corresponding to three types of forks—type 1 to 3, reminiscent of code clone classification [27, 30]—along a spectrum of increased sharing of artifacts commonly found in version control systems (VCS), such as commits and source code directories. The first definition, of type 1 forks, relies solely on code hosting platform information and requires no explicit VCS artifact sharing between repositories to be considered forks (although it allows it): Definition 2.1 (Type 1 fork, or forge fork). A repository 𝐵 hosted on code hosting platform 𝑃 is a type 1 fork (or forge fork) of repository 𝐴 hosted on the same platform, written 𝐴 ⇝1 𝐵, if 𝐵 has been created with an explicit “fork repository 𝐴” action on platform 𝑃. Although informal and seemingly trivial, this definition is both meaningful and actionable on current major code hosting platforms. For example, GitHub stores an explicit “forked from” relationship and makes it available via its repositories API:1 The parent and source objects are present when the repository is a fork. parent is the repository this repository was forked from, source is the ultimate source for the network.

Methodology:-

Our goal is to experimentally determine the amount and structure of forks for the various definitions we have introduced. To do so we will use two datasets: the Software Heritage Graph Dataset [25], which contains the development history needed to find intrinsic fork relationships, and a reference forge-specific dataset, GHTorrent [12], which contains the fork ancestry relationships as captured by GitHub GHTorrent. GitHub is the largest public software forge, and is therefore the candidate of choice to study forge forks (type 1). GHTorrent [12] crawls and archives GitHub via its REST API and makes periodical data dumps available in a relational table format. In its database schema, the project table contains a unique identifier for each repository, and a forked\_from column contains the ID of the repository it has been forked from if the repository is considered to be a forge forks. A single SQL query on this table allows to extract the full graph of GitHub-declared forks.

Fork networks:-

The easiest way to get a first sense of the amount and structure of forks according to the various definitions is to find all fork networks, as per Definition 2.4. This can be done in linear time with a simple graph traversal with linear complexity: two repositories are in the same network if and only if there exists a path between them in the undirected subgraph of origins and revisions. (We recall from the dataset section that we have removed the snapshot and revision layers, so that root commits are directly pointed by repository nodes.)

Fork cliques:-

While partitioning the corpus in fork networks gives a good idea of how intrinsic forks are linked together, it can group together repositories that are not forks of each other, as the intrinsic fork relationship is not transitive. Figure 6 shows a pattern, that we have verified as commonly found in the wild, where two different cliques will be merged in the same fork network—A and B are part of the same clique as they share development history; the same applies to B and C; whereas A and C do not share any part of their respective development histories but will end up in the same network. We expect this effect to merge cliques into giant components, that will make the size of the largest networks hard to interpret. The other interesting metric that can be looked at is the distribution of fork cliques, as defined in Definition 2.5. While cliques do not provide a partition function for the graph, they allow to narrow down the actual extent of forking relationships within large fork networks.

Result:-

We identified 71.9 M repositories in common between the Software Heritage Graph Dataset and GHTorrent, 41.4 M of which are nonempty. We focused our experiments on these repositories. different for shared root forks, where the average size is ≈ 10.5 and the frequency distribution is significantly farther from the reference distribution of forge forks. One distinguishing feature of each distribution of type 2 and type 3 forks is the size of the largest connected component, which is significantly larger than the largest networks of forge forks (by a factor of 17 for shared revision forks, and 157 for shared root forks). As discussed in Section 3.3, this is an expected outcome of our use of network as a quantification metric and confirms the need for further analysis through fork cliques. This does not however have any implications on the quantification aspect of the experiment, as partitioning this network further using fork cliques would still yield the same number of non-isolated repositories.

Conclusion:-

When relying only on forge-specific features and metadata to identify forked repositories, empirical studies on software forks might incur into selection and methodological biases. This is because repository forking can happen exogenously to any specific code hosting platform and out of band, especially when using distributed version control system (DVCS), which are currently very popular among developers. We also showed that the aggregation/merge dynamics into larger clusters of related repositories upon changing fork definitions is not just an absorption phenomenon into a “super attractor” cluster, but that it concerns all clusters: smaller ones are absorbed into larger ones of any size. The methodological implications of our findings are that: • Empirical software engineering studies on software forks aiming to be exhaustive in their coverage of forked repositories should consider using fork definitions based on shared VCS history rather than trusting forge-specific metadata. • Depending on the research question at hand, the objects of studies to consider when looking at repositories involved in forks are either fork networks or fork cliques. The latter have the advantage of excluding cases that exist in the wild (e.g., on GitHub) in which repositories that do not share VCS artifacts might end up in the same fork network due to transitiveness. • Any set of repositories can be partitioned in accordance with its relevant shared commit fork cliques by computing its fork p-clique partition function. This way of grouping together repositories that are all type 2 forks of each other is easily substitutable to partition approaches based on forge fork metadata.

Paper 1

Title And Authors:-

**An Empirical Study of Build Failures in the Docker Context**

* Yiwen Wu
* Yang Zhang
* Tao Wang
* Huaimin Wang

Conference Program ICSE 2020

Introduction And Motivation:-

Docker is one of the most popular containerization tools in current DevOps practice. It enables the encapsulation of software packages into containers and can run on any system . Since inception in 2013, Docker containers have been downloaded 130B+ times1 . The “Annual Container Adoption” report2 found that 79% of companies chose Docker as their primary container technology. With the widespread use and influence of Docker, many studies have been recently conducted to investigate its ecosystem configuration file. Those works have emerged a lot of great findings and brought many practical implications to developers, but were not designed to look into the details of Docker builds. Building is crucial to the software development process, which automates the process by which sources are compiled, linked, tested, packaged, and transformed into executable units.

Research And Methodology:-

• RQ1: (Frequency) How often do Docker builds fail?

• RQ2: (Fix effort) How long does it take to fix Docker build failures?

• RQ3: (Evolution) How do failures frequency and fix effort evolve across time?

The rest of this paper is organized as follows: Section 2 describes the study setup. Section 3 presents our study results. Section 4 outlines the research agenda and Section 5 discusses the threats to validity. Finally, Section 6 concludes the paper.

Motivation. The initial RQ aims at understanding how often Docker builds fail. Previous studies found a median of 37.4% of C++ builds and a 29.7% of Java builds at Google to be failing [7], and a 13.2% in the Visual Studio Context [5]. However, even though Docker is widely used in the open-source community [3], little research has been conducted on the Docker build failures. Thus, investigating the frequency of broken builds will help us to characterize the significance of build failures in the Docker context, and motivate the importance of our study. This RQ2 aims at understanding the time spent by developers fixing the Docker build failures. In the context of Google, developers spend a median of 5 and 12 minutes fixing C++ and Java build failures, respectively. This RQ aims at understanding whether the ratio (fix time) of broken builds is a constant value or fluctuates across time. Prior works [5, 7, 11] mainly focus on the static analysis of build failures, few studies have investigated the evolution of build failures, the only exception being the study by Zolfagharinia et al.

Result:- RQ1: Frequency analysis:-

In the Docker context, 152,897 of the 857,086 (17.8%) studied Docker builds fail, which is above the 11% as reported by Zhang et al. [11] (in the CI builds context) but below the 28.5% as reported by Seo et al. [7] (in the Java builds context). As discussed in the previous study [5], build failure rates are highly sensitive to changes in context. Further, we find that 3,260 (85.2%) projects have at least one broken build; 389 (10.2%) projects have at least 50% of broken builds. Thus, build failures are very common in the Docker context, and affect most of the open-source projects.

RQ2: Fix effort analysis:-

s the distribution of resolution time of broken Docker builds. Of the 10,566 valid Fail-Fix pairs, 511 (4.8%) pairs take less than 1 minute to fix; 2,105 (19.9%) pairs take 1 to 10 minutes to fix; 4,511 (42.7%) pairs take 10 to 100 minutes to fix; and 3,431 (32.5%) pairs take more than 100 minutes to fix. On average, broken builds in our dataset take an average of 124.5 minutes to fix (median time: 44.2 minutes), which is much longer than in Google’s study (less than 12 minutes) [7]. Many factors may contribute to this difference, e.g., work environments and tooling. Also, this large difference may be due to Google requiring developers to respond quickly to failures to avoid affecting a wide range of developers.

RQ3: Evolution analysis:-

w the distribution of failure ratio and fix time in different time periods, respectively. The red regression line shows the corresponding evolutionary trend over the eleven studied time periods (from 2014a to 2019a). We find that, the median build failure ratio decreases from 28.6% in 2014a to 15.8% in 2014b. While the median ratio increases across time from 16.7% in 2015a to 36.9% in 2019a. As for the fix time, the median build fix time decreases across time from 64 minutes in 2014a to 19 minutes in 2015a. While the median fix time increases across time from 22 minutes in 2015b to 97 minutes in 2019a. However, as the regression lines show, the overall build failure ratio and fix time have slightly increasing trends, especially when considering the logarithmic y-scale. These findings are very different from Zolfagharinia et al. [14], thus verifying that build failure rates and fix time are highly dependent on the study context.

Conclusion:- Executing the build process (i.e., building) is crucial to the software development process. Many prior studies have been published describing build failures in different contexts, to the best of our knowledge, this study is the first to conduct a large-scale empirical study of build failures in the Docker context. We quantify and analyze Docker build failures, including frequency, fix effort, and their evolution based on 857,086 Docker builds from 3,828 GitHub open-source projects. Our findings motivate the need for collecting more empirical evidence to better understand how developers build Docker containers and to guide future process refinements and tool development to improve Docker building efficiency.

Paper 3

Title And Authors:-

**The Secret Life of Commented-Out Source Code**

* Tri Minh Triet Pham
* Jinqiu Yang

Conference Program ICPC 2020

Introduction And Motivation:-

Source code comments play an important role in software development and maintenance, e.g., helping developers document and understand the code logics and ease the maintenance efforts out and modify buggy lines to produce temporary patches. Since developers may not be sure about the quality and stability of the fix, they may prefer to leave the original code logic in comments. Figure 2 shows an example of utilizing CO code in Apache CXF. When updated to work with JDK 11, line 3 was commented out as part of the fix for CXF-7741 that can quickly address the symptom of the unsuccessful build. Note that this temporary fix does not fix the root cause. Moreover, this temporary fix introduces flaky tests. The CO code was not uncommented until years later after several flaky tests were reported (i.e., CXF-8086 and CXF-8087). The CO code in this example is useful to help developers quickly decide how to fix the reported flaky tests. In total, we study CO code practices in six open-source projects of different sizes (LOC) and from diverse domains: VLC Android and NewPipe are two mobile applications from F-Droid dataset [3]; Groovy and CXF are desktop applications from Apache foundation [4]; JUNG is a popular network/graph framework that is used in a previous comment classification work [20]; and SWT is a graphical interface toolkit for Eclipse. CO code practice is controversial in the public eye. For example, we find many heated discussions on the usage of CO code on Stack Overflow While there exist valid concerns on CO code practice, there are cases where CO code practice may help developers effectively handle disruptions during the development. However, such effectiveness comes at a cost—developers may overlook the instability of CO code and forget to stabilize the temporary solutions that use CO code. using the keyword “commented out code”.

Research And Methodology:-

• RQ1: How prevalent is CO code practice?

• RQ2:What are the evolution patterns of CO code?

• RQ3: How often do CO code lines co-transit?

• RQ4: What are the purposes of utilizing CO code practice?

**RQ1: How prevalent is CO code practice?**

s the total number of comment lines, the numbers of comment lines in license and Javadoc, and the total number of CO code lines for the studied projects. The majority of the comments are about license and Javadoc. Only a small portion of all the comments are code comments that explain code logics using natural language. Table 3 also shows the percentage of CO code lines among all the comments, ranging from 1.2% to 5.3%. After excluding license and Javadoc, the percentage of CO code lines in the studied projects ranges from 2.3% to 9.5%. The results shows that CO code instances are not prevalent in the recent versions. In addition, we also analyze the CO code activities in different development periods in the entire development history. Figure 4 shows that how many code lines are being actively developed/maintained in the equally divided ten time intervals. In particular, we quantify the development/maintenance activities using two metrics: 1) how many CO code lines are being added (the white bars in Figure 4), i.e., by either the transitions comment or introduce; and 2) how many CO code lines are no longer commented out (the gray bars in Figure 4), i.e., by either the transitions uncomment or fade.

**RQ2: What are the evolution patterns of CO code?**

s the details of CO code evolution patterns in the studied projects. Most of the CO code instances will never become live code again, i.e., P1 and P2 take the majority of the cases. Most of Pattern 1 cases are caused by commenting out code before deletion, usually as part of a large-scale refactoring or implementation change, e.g., a to-be-implemented feature is significantly modified. Pattern 2 means that the CO code instances remain in the repositories as of the newest version. They may become live code in the future. The P2 cases are typically caused by incomplete features that developers are still actively working on, e.g., the relevant test cases are commented out to avoid meaningless failures. The results show that developers leverage CO code practice to temporarily harbor some code logics which will later be reenabled. In fact, the utilization of code logics in CO code may be more frequent than what the union of P3 and P4 indicates. Recall the example shown in Introduction (Figure 1), CO code line may become obsolete but its code logic can still be reused by developers.

**RQ3: How often do CO code lines co-transit?**

s how often CO code lines co-transit, which is reflected by uncomment rate of the relevant follow-up commits. In particular Figure 6 highlights the percentages of commits with uncomment rate of 100% and non-100%. As we analyzed two types of transitions, we present the results separately, i.e., the CO code lines that are 1) introduced as CO code → uncommented (the left side) and 2) commented out → uncommented (the right side). A non-100% uncomment rate indicates that the involved CO code lines are more scattered in the evolution, i.e., it may take several fellow-up commits to uncomment all the CO code lines that are initially introduced in the same commit. If the evolution of CO code instances is more scattered, it would be more challenging to perform uncommenting activities since the ad-hoc and tedious process may be more error-prone. Our study shows that many of the CO code-related commits are scattered, e.g., 87% of the relevant commits in Eclipse SWT involving gradually uncommenting part of the CO code lines. One reason we notice is that the total number of CO code introduced directly may contain a larger number of CO code lines, e.g., typically for implementing new features. An example of such practice is that developers add a chunk of code as CO code to implement a new feature, which is not completely ready due to various reasons, hence developers may need to gradually uncomment the CO code lines.

**RQ4: What are the purposes of utilizing CO code practice?**

Our manual study (i.e., on 342 comment transitions and 161 uncomment transitions) reveals a list of motivations behind the transitions on CO code. When CO code is introduced as CO code (i.e., one type of comment transition), the most common motivation is to introduce new functionalities, however developers temporarily comment out such code since the dependent functionalities are not complete as a result of work in progress, pending refactoring or design changes. An interesting exception is that, in VLC Android we find a large number of CO code instances introduced due to importing the source code of an external library into the VLC codebase. Some other common usages include copying code templates from documentation that have CO code embedded, and for code debugging. The transition of faded as CO code is very straightforward, for which the primary purpose we find is to remove CO code and clean up codebase. For the transition of commented, the most common motivations include marking the code for later permanent removal during nontrivial software maintenance efforts, such as refactoring, adding new functionalities, or design change. For the transition of uncomment, there are two primary motivations. First, similar to the motivation of introduced as CO code, developers uncomment CO code to introduce the functionalities in CO code when the roadblocks are removed, i.e., the dependent code is ready. Second, we find that developers often uncomment CO code to revert the changes made by previous commits in which developers comment out source code. We also notice that developers may uncomment CO code both intentionally and unintentionally. Developers intentionally uncomment CO code to test new functionality, to toggle features, or to fix bugs.

Result:-

In this section, we summarize prior studies on comment and source code analysis, self-admitted technical debt and feature toggling. Comments quality assessment and categorization. There have been many studies on comment analysis. Here we describe the most relevant studies that focuses on discussing the quality of comment and performing comment categorization. Haouari et al. [7] proposed a taxonomy of comments that includes categories in total. The categorization is based on four dimensions, i.e., object, type, style and quality. Steidl et al. [20] proposed a quality model and metrics to assess the quality of comments and surveyed to show the relevance of their metric to developers in practice. In particular, they proposed to use classification techniques to categorize comments automatically based on a mixture of location and purpose. Their classification approach achieves a precision of 89% and a recall of 95% for identifying CO code. Pascarella and Bacchelli [16] extended the work by Steidl et al. [20] by providing finer-grained comment categories. In addition, Pascarella and Bacchelli experimented different classification algorithms and showed promising results to classify comments into six major categories and 16 sub-categories.

Conclusion:-

Commented-out code is cited as a controversial coding practice. Despite the popularity of studies on comments, CO code is often excluded in previous studies. In this paper, we conduct the first study on CO code and focus on its prevalence, evolution, motivation, and potential maintenance challenges. We develop automated solutions to detect CO code instances and track their histories in code repositories. Moreover, we perform a manual analysis to understand the adoption of CO code practices and the related challenges. Our study on six open-source software systems reveals that despite the low prevalence of CO code in a recent version, CO code practice is actively used in certain development phases, i.e., up to 20% of the commits involving at least one CO code instance. Moreover, by analyzing the evolution of CO code, we find that while CO code introduced in the same commit does not often co-transit again, e.g., being uncommented together. Last, our manual analysis shows that there exist various maintenance reasons for developers to utilize CO code practice. This paper presents insights on CO code practice that is previously overlooked, and sheds light on its prevalence and evolution in software development. Further studies are needed to provide tooling support to help developers better maintain and utilize CO code.